Processes propose values and must agree on a common decision value, proposed by some process.
Consensus, More Formally

Each process $p_i$ proposes a binary input value, $x_i$, and returns an output value (decision), $y_i$.

Only two conditions on the output:
- **Agreement**: all processes decide on the same value
- **Validity**: this value is the proposal of some process

Consensus is Easy...

If processes can wait for each other.

```plaintext
propose(v)
   R[i] = v
   while true
      read R[1,...,n] to l[1,...,n]
      if all l[1,...,n] ≠ ⊥
         decide on min l[1,...,n]
         (and return)
```
Termination (Liveness)

**Solo-termination** (also called **obstruction-freedom**): A process has to terminate if (eventually) it runs by itself.

**Nonblocking**: From any configuration, if some process takes infinitely many steps, then some process (not necessarily the same one) terminates, regardless of steps by other processes. (Analogue of no deadlock for mutex.)
Termination (Liveness)

**Solo-termination** (also called obstruction-freedom): A process has to terminate if (eventually) it runs by itself.

**Nonblocking**: From any configuration, if some process takes infinitely many steps, then some process (not necessarily the same one) terminates, regardless of steps by other processes.

**Wait-freedom**: From any configuration, if some process takes infinitely many steps, then the process terminates, regardless of steps by other processes. (Analogue of no starvation for mutex.)

Consensus with Compare&Swap

Wait-free consensus for any number of processes using compare&swap is easy

Use a single shared variable, `first`, initially ⊥

```plaintext
propose(x)
    v = cas(first, ⊥, x)
    if v == ⊥ decide x
    else decide v
```

What happens if we use only reads and writes?
Graded Consensus (Adopt-Commit)

Like consensus but the decision is \((\text{grade}, y_i)\), grade is either \textit{adopt} or \textit{commit}, such that

\textbf{Graded agreement}: if a process decides \((\text{commit}, y_i)\) then all processes decide either \((\text{adopt}, y_i)\) or \((\text{commit}, y_i)\)

\textbf{Validity}: \(y_i\) was proposed by some process

\textbf{Convergence}: If only \(y_i\) is proposed before \(p\) outputs \((\text{grade}, y_i)\) then \textit{grade} = \textit{commit}

- Two special cases: (a) \(p\) runs alone (b) same proposals
Wait-Free Adopt-Commit Provides Solo Terminating Consensus

Use infinitely many copies of adopt-commit₁, …

```
propose(v)
  for ( m = 1; ; m++)
    (grade,value) = adopt-commitₘ(v)
    if grade == commit return value
    else v = value
```

Validity and agreement are immediate from the related properties of the adopt-commit protocol
Solo termination follow from convergence (a)

Adopt-Commit with SW Registers

Two arrays of single-writer variables A[1,…,n] B[1,…,n], all initially ⊥

```
adopt-commit(v)
  write v to A[i]
  read A[1],…,A[n] // one by one
  if only one non-⊥ value w, B[i] = “commit w”
  else B[i] = “adopt v”
  read B[1],…,B[n] // one by one
  if only “commit w”, return (commit,w)
  else if contains “commit w”, return (adopt,w)
  else return (adopt,v)
```

must be v
Proof of SW Adopt-Commit

✓ Validity is clear
✓ Convergence follows from inspecting the code

```
adopt-commit(v)
  write v to A[i]
  read A[1],…,A[n] // one by one
  if only one non-⊥ value w, B[i] = “commit w”
  else B[i] = “adopt v”
  read B[1],…,B[n] // one by one
  if only “commit w”, return (commit,w)
  else if contains “commit w”, return (adopt,w)
  else return (adopt,v)
```

Graded Agreement of SW Adopt-Commit

Lemma: If \( p_i \) writes “commit \( v \)” to \( B[i] \), then no process writes “commit \( w \)” to \( B \), \( w \neq v \)

\( v \) must be the first value written in the array \( A \)
If \( p_i \) returns (commit,v), then “commit \( v \)” is the first value written in \( B \)
\( \Rightarrow \) all processes commit or adopt \( v \)

```
adopt-commit(v)
  write v to A[i]
  read A[1],…,A[n] // one by one
  if only one non-⊥ value w, B[i] = “commit w”
  else B[i] = “adopt v”
  read B[1],…,B[n] // one by one
  if only “commit w”, return (commit,w)
  else if contains “commit w”, return (adopt,w)
  else return (adopt,v)
```
**Adopt-Commit with MW Registers**

shared variables:
- Proposal, initially ⊥
- $R_0$, $R_1$, initially ⊥
- local variable preference

```plaintext
adopt-commit(v)
    R_v = 1
    if Proposal ≠ ⊥ then preference = Proposal
    else
        preference = v
        Proposal = preference
        if $R_{1-v} ≠ ⊥$ then return (adopt, preference)
        else return (commit, preference)
```

---

**What about Wait-Free Consensus?**

Nonblocking consensus implies wait-free consensus (using a mw register R)

- Execute the nonblocking consensus
- After deciding, write decision to R
- Interleave reading R with nonblocking consensus
What about Wait-Free Consensus?

- Wait-free consensus is impossible
  - Even if we can wait for all but one process

- Implies the same for nonblocking consensus
- Holds for message passing and shared memory

Fischer  Lynch  Patterson

Potence of a Configuration C

- **C is *v*-potent** if *v* is decided in some configuration reachable from *C*
Valence of a Configuration $C$

- $C$ is **v-potent** if $v$ is decided in some configuration reachable from $C$

- $C$ is **v-univalent** if it is $v$-potent but not $(1-v)$-potent (univalent in general)

- $C$ is **bivalent** if it is both 0-potent and 1-potent

Univalent Similarity

**Lemma:** If $C_1$ and $C_2$ are both univalent and they are similar w.r.t. some process $p_i$, then they have the same valence.

**Proof:**

By wait-freedom

$v \equiv w$
Impossibility of Two-Process Consensus using Reads / Writes

**Proof overview:** If there is a 2-process wait-free consensus algorithm

- Show there is a bivalent initial configuration
- Show that from every bivalent configuration there is an execution leading to a bivalent configuration

\( \Rightarrow \) No process decides

For simplicity assume single-writer variables

There is a Bivalent Initial Configuration

Assume all initial configurations are univalent
There is a Bivalent Initial Configuration

Assume all initial configurations are univalent
These configurations look the same to \( p_1 \)

\[(0,1) \text{ p}_1\text{-only} \rightarrow 0 \]
\[(1,1) \text{ p}_1\text{-only} \rightarrow 0 \]

Contradiction!

Consider the configurations reachable by a single step of each process
If either of them is bivalent \( \Rightarrow \) we are done
Both are univalent \( \Rightarrow \) 1-valent & 0-valent

Lemma: If \( C \) is bivalent then a bivalent configuration \( C' \) is reachable from \( C \).
Extending a Bivalent Configuration, One Step at a Time

**Lemma:** If $C$ is bivalent, then a bivalent configuration $C'$ is reachable from $C$.

Consider the configurations reachable by a single step of each process. If either of them is bivalent $\implies$ we are done. Both are univalent $\implies$ 1-valent & 0-valent.

---

Extending a Bivalent Configuration, Steps that Commute

**Lemma:** If $C$ is bivalent, then a bivalent configuration $C'$ is reachable from $C$.

Case 1: both read or both write (different variables). Their steps commute.

Contradiction!
Extending a Bivalent Configuration, Overwriting Step

**Lemma:** If C is bivalent then a bivalent configuration C’ is reachable from C.

Case 2: p₀ reads, p₁ writes (or vice versa) to the same variable

Covering...

Contradiction!
The Full Impossibility Result

• In an asynchronous system, consensus cannot be solved when the algorithm has to tolerate **even just a single failure**
  – n-1 processes cannot take an infinite number of steps without deciding
• Holds for the shared-memory model **as well as for the message-passing model**
• Describe both proofs in a unified manner using **layered executions**
  – below, f is the number of processes that may fail, we concentrate on the case f = 1

Layered Schedules

**f-layer**: sequence of at least n-f different processes
  – Order is sometimes important
**f-schedule**: sequence of f-layers

- p_2 is **faulty** in layer 2, but nonfaulty in layer k
- p_n **crashes** in layer 3 – faulty in every layer r, 3 ≤ r ≤ k

An f-schedule \( \sigma \) and an initial configuration I determine a **layered execution** \( \alpha(\sigma, I) \)
Similarity

\[ \alpha_1 \overset{p}{\sim} \alpha_2 \]

In execution $\alpha_1$

\[
\begin{array}{ccc}
  I_1 & p_1 & p_2 & \ldots & p_n \\
  \quad \text{layer 1} & & & & \\
  I_2 & p_1 & p_2 & \ldots & p_{n-f} \\
  \quad \text{layer 2} & & & & \\
  I_k & p_1 & p_2 & \ldots & p_1 \\
  \quad \text{layer k} & & & & \\
\end{array}
\]

$p$: decide $v$

In execution $\alpha_2$

\[
\begin{array}{ccc}
  I_1 & p_1 & p_2 & \ldots & p_n \\
  \quad \text{layer 1} & & & & \\
  I_2 & p_1 & p_2 & \ldots & p_{n-f} \\
  \quad \text{layer 2} & & & & \\
  I_k & p_1 & p_2 & \ldots & p_1 \\
  \quad \text{layer k} & & & & \\
\end{array}
\]

$p$: decide $v$

Connectivity

\[ \alpha_1 \approx \alpha_m \equiv \alpha_1 \overset{p}{\sim} \alpha_2 \overset{p'}{\sim} \alpha_3 \overset{q}{\sim} \ldots \overset{q}{\sim} \alpha_m \Rightarrow \text{same decision} \]

\[
\begin{array}{ccc}
  I_1 & p_1 & p_2 & \ldots & p_n \\
  \quad \text{layer 1} & & & & \\
  I_2 & p_1 & p_2 & \ldots & p_{n-f} \\
  \quad \text{layer 2} & & & & \\
  I_k & p_1 & p_2 & \ldots & p_1 \\
  \quad \text{layer k} & & & & \\
  I_m & p_1 & p_2 & \ldots & p_1 \\
  \quad \text{layer k} & & & & \\
\end{array}
\]

$p$: decide $v$

\[ \alpha_1 \overset{p}{\sim} \alpha_2 \]

$p,p'$: decide $v$

\[ \alpha_2 \overset{p'}{\sim} \alpha_3 \]

$p',p''$: decide $v$

\[ \alpha_{m-1} \overset{q}{\sim} \alpha_m \]

$q$: decide $v$
Key Lemma: Crashing a Process

\[ \text{crash}(\sigma, p, r): \text{p crashes in layer r of } \sigma \]

\[ \begin{array}{c}
\begin{array}{cccc}
\text{p}_1 & \text{p}_2 & \ldots & \text{p}_s \\
\text{layer 1} \\
\end{array} & \begin{array}{c}
\begin{array}{cccc}
\text{p}_1 & \text{p}_2 & \ldots & \text{p}_8 \\
\text{layer 2} \\
\end{array} & \ldots & \begin{array}{c}
\begin{array}{cccc}
\text{p}_8 & \text{p}_2 & \ldots & \text{p}_1 \\
\text{layer k} \\
\end{array} \\
\end{array}
\end{array}
\end{array} \]

\[ \text{crash}(\sigma, p_8, 2) \]

Lemma: For every input configuration \( I \), f-schedule \( \sigma \), process \( p \) and round \( r \), 
\[ \alpha(\sigma, I) \approx \alpha(\text{crash}(\sigma, p, r), I) \]

Deriving the Impossibility Result: Input Connectivity

Consider a sequence of input configurations
\[ I_0 = (0,0,\ldots,0) \]
\[ I_1 = (1,0,\ldots,0) \]
\[ I_2 = (1,1,\ldots,0) \]
\[ \ldots \]
\[ I_n = (1,1,\ldots,1) \]

Claim: \[ \alpha(\sigma_F, I_0) \approx \alpha(\sigma_F, I_n) \], where \( \sigma_F \) is the full layered schedule

⇒ Same decision in \( I_0 \) (all zeroes) and \( I_n \) (all ones)

Contradiction!
Proving the Key Lemma

Lemma: For every input configuration $I$, process $p$, and round $r$, $\alpha(\sigma, I) \approx \alpha(\text{crash}(\sigma, p, r), I)$

The proof is very model dependent
- Shared memory: read / write (single-writer) ✓
- Message passing

Need to assume bounded executions
swap(σ,p_i,r)

Process p_i is swapped with the next process (p_j) in layer r

Both read or write (different registers) ⇒ no process distinguishes. 

p_j reads and p_i writes ⇒ only p_j distinguishes at the end of layer r
delay(σ, pᵢ, r)

The idea is to crash pᵢ by swapping every appearance of pᵢ until after the end of the schedule. But cannot swap with another appearance of pᵢ. Delay pᵢ by one layer, starting from layer r.

Swaps + Delays $\Rightarrow$ Crash

Now continue swapping...
Algebraically

\[ \text{delay}(\sigma, p, r) = \text{swap}^k(\text{rollover}(\text{swap}^{k'}(\text{delay}(\sigma, p, r+1), p, r), p, r), p, r+1) \]

\( k' \) is \( p \)'s distance from the end of layer \( r \)
\( k \) is \( p \)'s distance from the beginning of layer \( r+1 \)

Similarly

\[ \text{crash}(\sigma, p, r) = \text{crash}(\text{delay}(\sigma, p, r), p, r+1) \]

Impossibility Result for Message-Passing Systems

Original context of this result (FLP)
Original proof has a different structure (similar to previous lecture)
Message-passing: Model of Computation

In each step, send messages to all processes
In layered executions, we synchronize the steps

Message-passing

• Crash $p_i$ by removing it from all layers
  – Incremental $\Rightarrow$ remove messages from $p_i$ to $p_j$
  – Inductively, crash $p_j$ in following layers
• Repeat for all layers $\Rightarrow p_i$ crash
Bounding the Executions

• Why?
  • To have a well-defined base case for the (backwards) induction on the layer number

• How?
  • The proof considers a fixed (and bounded) set of executions from n+1 input configurations

Consensus in Synchronous Systems

“Then we are agreed nine to one that we will say our previous vote was unanimous!”
Synchronous Systems

• Processes take steps in **rounds**
• In each round, a process
  – sends messages to all (other) processes
  – receives messages from all other processes
  – does some local computation

Crash Failures in Synchronous Systems

• All but at most \( f \) **faulty processes** take an infinite number of steps (or until everyone decides)
• Once a faulty processor fails to take a step in a round, it takes no more steps
• In the last step of a faulty process, some subset of its outgoing messages are sent
Consensus Algorithm for Crash Failures

• Tolerates $f < n$ crash failures
• Requires $f + 1$ rounds

Consensus Algorithm for Crash Failures

Each process executes the following code

```
v = my input  
in each round 1 through f+1:  
    if v not sent before, send v to all  
    wait to receive messages for this round  
    v = min of received values and current value of v  
in round f+1, decide on v
```

• Tolerates $f < n$ crash failures
• Requires $f + 1$ rounds
• A total of $\leq n^2/|V|$ messages each with $\log |V|$ bits, where $V$ is the input set.
An Execution of the Algorithm:
$p_i$ with input $v_i$

$v = my \ input$

in each round 1 through $f+1$:
- if $v$ not sent before, send $v$ to all
- wait to receive messages for this round
- $v = \min$ of received values and current value of $v$ in round $f+1$, decide on $v$

• round 1:
  - send input
  - receive round 1 messages
  - compute value for $v$

• round 2:
  - send $v$ (if this is a new value)
  - receive round 2 messages
  - compute value for $v$

Correctness of Crash Consensus Algorithm

**Termination:** By the code, finish in round $f+1$.

**Validity:** processes do not create values.
If all inputs are the same, then that is the only value ever sent around (and decided)
Crash Consensus Algorithm: Agreement

Suppose in contradiction $p_j$ decides on a smaller value, $x$, than $p_i$ does

$\Rightarrow x$ was hidden from $p_i$ by a chain of faulty processes (one for each round)

$\Rightarrow$ This chain has $f + 1$ faulty processors, a contradiction

Is this the Best Round Complexity?
Rounds Lower Bound: Initial Lemma

Lemma: From some initial configuration, there are two executions $\gamma$ and $\alpha$, in which two different values are decided. $\gamma$ is failure-free, and in $\alpha$, one process crashes before taking any steps, but no other processes fail.

\begin{align*}
C_0 &= (0,0,\ldots,0,0) & v_0 &= 0 \text{(by validity)} \\
C_j &= (0,0,\ldots,1,1) & \text{failure-free} & v_i \text{ is decided} \\
C_n &= (1,1,\ldots,1,1) & v_n &= 1 \text{(by validity)}
\end{align*}

Rounds Lower Bound: Proof of Initial Lemma

Lemma: From some initial configuration, there are two executions $\gamma$ and $\alpha$, in which two different values are decided. $\gamma$ is failure-free, and in $\alpha$, one process crashes before taking any steps, but no other processes fail.

\begin{align*}
C_j &= (0,0,\ldots,0,1) & \text{failure-free} & 0 \text{ is decided} \\
p_j \text{ crashes at the start} & 0 \text{ is decided?} \\
C_{j+1} &= (0,0,\ldots,1,1) & \text{failure-free} & 1 \text{ is decided}
\end{align*}
Rounds Lower Bound: Proof of Initial Lemma

**Lemma:** From some initial configuration, there are two executions $\gamma$ and $\alpha$, in which two different values are decided. $\gamma$ is failure-free, and in $\alpha$, one process crashes before taking any steps, but no other processes fail.

$C_j = (0,0,...,0,1)$

- $p_j$ crashes at the start
- $1$ is decided

$C_{j+1} = (0,0,...,1,1)$

- failure-free
- $0$ is decided

Rounds Lower Bound: Main Lemma

We consider only $f$-round executions such that:
- $f \leq n-2$
- At most one process crashes in each round and at most $f$ processes crash in each execution.
- In the round in which a process crashes, it sends messages to a prefix of processes, ordered by id’s

**Lemma:** For any $f$-round execution $\alpha$, $\alpha \approx \gamma$, where $\gamma$ is the same as $\alpha$ during the first $r$ rounds but has no crashes after round $r$, $0 \leq r \leq f$. 
Rounds Lower Bound: 
Proof of Main Lemma (Base)
By backward induction on $r$

The base case, $r = f$, $\alpha = \gamma$ and the lemma is obvious

**Lemma:** For any $f$-round execution $\alpha$, $\alpha \approx \gamma$, where $\gamma$ is the same as $\alpha$ during the first $r$ rounds but has no crashes after round $r$, $0 \leq r \leq f$.

Rounds Lower Bound: 
Proof of Main Lemma (Inductive Step)
Assume $r < f$ and that the lemma holds for $r+1$. 

![Diagram](image)
Rounds Lower Bound: Proof of Main Lemma (Inductive Step)

Assume $r < f$ and that the lemma holds for $r+1$. Let $\beta$ be the same as $\alpha$ during its first $r+1$ rounds and has no crashes after round $r+1$. By induction, $\alpha \approx \beta$; we need to show $\beta \approx \gamma$.

What happens in $\beta$?

$p$ is the single process that crashes in round $r+1$ of $\beta$ (if none fails then we are done).

$q_1, ..., q_t$ are the correct processes to which $p$ does not send a message in round $r+1$ (in order of id’s).
Rounds Lower Bound: Chain of Executions

$\beta_k$ is the same as $\beta$ in the first $r+1$ rounds, except that $p$ sends messages to $q_1, \ldots, q_k$ in round $r+1$

$\beta_0 = \beta$

A correct process does not distinguish $\beta_t$ from $\gamma$

Rounds Lower Bound: $r = f-1$

Some correct process $\neq q_k$ does not distinguish between $\beta_k$ and $\beta_{k-1}$ (there is one since $f < n-2$)

$\Rightarrow \beta \approx \beta_t \approx \gamma$
Rounds Lower Bound:
\[ \beta_k \approx \beta_{k-1} \text{ for } r < f-1 \]

\( \gamma_k \) is the same as \( \beta_k \) for the first \( r+1 \) rounds, but \( q_k \) crashes in the beginning of round \( r+2 \) (cleanly) and there are no crashes after round \( r+2 \). By induction, \( \beta_k \approx \gamma_k \)

\( \gamma_k' \) is the same as \( \beta_{k-1} \) for the first \( r+1 \) rounds, but \( q_k \) crashes in the beginning of round \( r+2 \) (cleanly) and there are no crashes after round \( r+2 \). By induction, \( \beta_{k-1} \approx \gamma_k' \)

\[ \gamma_k \approx \gamma_k' \Rightarrow \beta_k \approx \beta_{k-1} \]

Completing the Proof

**Theorem:** Any consensus algorithm for \( n \geq f+2 \) processes that tolerates \( f \) crashes requires \( \geq f+1 \) rounds

Otherwise, apply initial configuration lemma

There is an initial configuration from which there are two executions \( \alpha \) and \( \gamma \) that decide different values

In \( \alpha \) and \( \gamma \) no processes crashes, except for one process that crashes before the start of \( \gamma \)

By previous lemma, \( \alpha \approx \gamma \)

\( \Rightarrow \) Same value is decided in both
Byzantine Failures

How Many Processes can Solve Consensus with One Byzantine Failure?

Validity: If all nonfaulty processes have input \( v \), decide \( v \)

- Two processes?
  If \( p_0 \) has input 0 and \( p_1 \) has 1, someone has to change, but not both

  What if one processor is faulty?
  How can the other one know?

- Three processes?
  If \( p_0 \) has input 0, \( p_1 \) has input 1, and \( p_2 \) is faulty, then a tie-breaker is needed, but \( p_2 \) can act maliciously
# Processes Lower Bound for $f = 1$

**Theorem:** Any consensus algorithm for one Byzantine failure must have at least four processes

Suppose in contradiction there is a consensus algorithm for 3 processes and 1 Byzantine failure

Get two copies

Rewire the copies
# Processes Lower Bound for $f = 1$

Rewire the copies and assign inputs
This execution does not have to solve consensus
But it can specify the behavior of faulty processes
# Processes Lower Bound for f = 1

### Diagram 1

- $p_0$ and $p_1$ must decide 1
- $p_0$ and $p_1$ must decide 1
- $p_1$ and $p_2$ must decide 0
- $p_1$ and $p_2$ must decide 0
- $p_2$ acts as if it has 0
- $p_2$ acts as if it has 1

### Diagram 2

- $p_0$ and $p_2$ must agree
- $p_0$ and $p_1$ must decide 1
- $p_1$ and $p_2$ must decide 0
- $p_0$ and $p_2$ must decide 0
- $p_1$ and $p_2$ must decide 0
- $p_2$ acts as if it has 0
- $p_2$ acts as if it has 1
n > 3f for arbitrary f

**Theorem:** Any consensus algorithm for $f$ Byzantine failures must have at least $3f+1$ processes

Proof by reduction to the 3:1 case

- Suppose in contradiction there is an algorithm $\mathcal{A}$ for $f > 1$ failures and $n = 3f$ total processes
- Use $\mathcal{A}$ to construct an algorithm for 1 failure and 3 processors, a contradiction

---

The Reduction

Partition the $n \leq 3f$ processes into three sets, $Q_0$, $Q_1$, and $Q_2$, each of size at most $f$

- $p_0$ simulates $Q_0$
- $p_1$ simulates $Q_1$
- $p_2$ simulates $Q_2$

If one process is faulty in the $n = 3$ system, then at most $f$ processes are faulty in the simulated system

$\Rightarrow$ The simulated system is correct

Processes in the $n = 3$ system decide as the simulated processes

$\Rightarrow$ Their decisions are correct
Tree Algorithm

- This algorithm uses
  - $f + 1$ rounds (optimal)
  - $n = 3f + 1$ processors (optimal)
  - exponential size messages (very bad)
- Each process keeps a local tree data structure
- Values are filled in the tree during the $f + 1$ rounds
- Then, the decision is calculated from the tree values

Local Tree Data Structure

Each node is labeled with a sequence of unique process identifiers.
Root's label is the empty sequence $\lambda$; its level is 0.
Root has $n$ children, labeled 0 .. $n - 1$.

For $n = 4$, $f = 1$:
Local Tree Data Structure

Child node labeled \( i \) has \( n - 1 \) children, labeled \( i : 0 \ldots i : n-1 \) (skipping \( i : i \))

Node at level \( d \) with label \( v \) has \( n - d \) children, labeled \( v : 0 \ldots v : n-1 \) (skipping any index in \( v \))

Nodes at level \( f + 1 \) are **leaves**

\[ n = 4, \, f = 1 \]

Filling in the Tree Nodes

- Initially store your input in the root (level 0)
- Round 1:
  - send level 0 of your tree to all
  - store value \( x \) received from each \( p_j \) in tree node labeled \( j \) (level 1); use a default if necessary
  - "\( p_j \) told me that \( p_j \)'s input is \( x \)"
- Round 2:
  - send level 1 of your tree to all
  - store value \( x \) received from each \( p_j \) for each tree node \( k \) in tree node labeled \( k : j \) (level 2); use a default if necessary
  - "\( p_j \) told me that \( p_j \)'s input is \( x \)"
- Continue for \( f + 1 \) rounds
Example Execution: Round 1

Tree at p₀

0 1 2 3
1 0 1 0

Tree at p₁

0 1 2 3
0 1 2 1

Example Execution: Round 2

Tree at p₀

0 1 2 3
1 0 1 0

Tree at p₁

0 1 2 3
0 1 2 3

© Hagit Attiya 2019-20
Calculating the Decision

• In round $f + 1$, each process uses the values in its tree to compute its decision
• Recursively compute $\text{resolve}(\lambda)$ for the root, based on the "resolved" values for its children

\[
\begin{align*}
\text{resolve}(\pi) &= \begin{cases} 
\text{value in tree node labeled } \pi \text{ if it is a leaf} \\
\text{majority}\{\text{resolve}(\pi') : \pi' \text{ is a child of } \pi\} \\
0 \quad \text{if no majority (use a default if tied)}
\end{cases} \\
\end{align*}
\]

$n = 4$, $f = 1$
Resolved Values are Consistent

**Lemma:** If $p_i$ and $p_j$ are nonfaulty, then $p_i$’s resolved value for tree node labeled $\pi j$ (what $p_j$ tells $p_i$ for node $\pi$) equals what $p_j$ stores in its node $\pi$

Proof by induction $\pi$’s height (starting at the leaves)

By inductive hypothesis, resolved values for $\pi$ children corresponding to nonfaulty processes are consistent

Since $n > 3f$ and $\pi$ has $\geq n - f$ children majority of children correspond to nonfaulty processes

Resolved Values are Valid

- Suppose all nonfaulty processes have input $v$
- Nonfaulty process $p_i$ decides $\text{resolve}(\lambda_i)$, which is the majority among $\text{resolve}(j)$, $0 \leq j \leq n-1$, based on $p_i$’s tree
- Since resolved values are consistent, $\text{resolve}(j)$ (at $p_j$) is the value stored at the root of $p_j$’s tree, which is $p_j$’s input value if $p_j$ is nonfaulty
- Since there is a majority of nonfaulty processes, $p_i$ decides $v$
Common Nodes

A tree node $\pi$ is **common** if all nonfaulty processes compute the same value of resolve($\pi$)

$\pi$'s tree

$\pi_j$'s tree

*same resolved value*

Common Frontiers

A tree node $\pi$ has a **common frontier** if there is a common node on every path from $\pi$ to a leaf

common nodes
Common Nodes and Frontiers

**Lemma:** If $\pi$ has a common frontier, then $\pi$ is common

Proof by induction on height of $\pi$, since resolve uses majority

Implies **agreement:**

- On each root-leaf path there is at least one node corresponding to a nonfaulty process
  - The nodes on the path correspond to $f + 1$ different processes
  - There are at most $f$ faulty processes
  $\Rightarrow$ This node is common (by consistency of resolved values)
  $\Rightarrow$ The root has a common frontier
  $\Rightarrow$ The root is common

Complexities of the Tree Algorithm

- $n > 3f$ processors
- $f + 1$ rounds
- exponential size messages:
  - each message in round $r$ contains $n(n-1)(n-2)...(n-(r-2))$ values
  - When $r = f + 1$, this is exponential if $f$ is more than a constant relative to $n$
A More Efficient Algorithm?

Better message complexity by increasing the number of rounds and ratio of nonfaulty processes

- $n > 4t$, $2f + 1$ rounds

Aside: there are algorithms with

- Polynomial number of message bits
- $f + 1$ rounds
- $n > 3t$

### Phase King Algorithm

$(n > 4t, 2(f+1) \text{ rounds})$

Code for process $p_i$

```plaintext
pref = my input

**first round of phase $k$, $1 \leq k \leq f+1$:**
- send pref to all
- receive prefs of others
- let maj be value that occurs > $n/2$ times // default 0
- let mult be number of times maj occurs

**second round of phase $k$:**
- if $i = k$ then send maj to all // I am the phase king
- receive tie-breaker from $p_k$ // default 0
- if mult > $n/2 + f$ then
  - pref := maj
- else
  - pref := tie-breaker
- if $k = f + 1$ then decide pref
```
Unanimous Phase Lemma

**Lemma:** If all nonfaulty processes prefer $v$ at start of phase $k$, then all prefer $v$ at end of phase $k$

Since $n > 4f$, it follows that $n - f > n/2 + f$

Therefore, if all nonfaulty processes have input $v$:
- At start of phase 1, all nonfaulty processes prefer $v$
- At end of phase 1, all nonfaulty processes prefer $v$
- At start of phase 2, all nonfaulty processes prefer $v$
- At end of phase 2, all nonfaulty processes prefer $v$
- ...
- At end of phase $f + 1$, all nonfaulty processes prefer $v$ and decide $v$

Nonfaulty King Lemma

**Lemma:** If $p_k$ is nonfaulty, then all nonfaulty processes have same preference at end of phase $k$

**Proof:** If two nonfaulty processes $p_i$ and $p_j$ use $p_k$'s tie-breaker, they have same preference.

If $p_i$ uses a majority value $v$ and $p_j$ uses $p_k$'s tie-breaker then $p_k$ majority value is also $v$

If both $p_i$ and $p_j$ use their majority value, then it must be the same value.
Agreement in Phase King Algorithm

\( f + 1 \) iterations \( \Rightarrow \) at least one with a nonfaulty king

Nonfaulty King Lemma \( \Rightarrow \) at the end of that phase, all nonfaulty processes have same preference

Unanimous Phase Lemma \( \Rightarrow \) from that phase on, all nonfaulty processes have same preference

\( \Rightarrow \) All nonfaulty processes decide on the same value

Phase Queen Algorithm

\((n > 3t, 3(f+1) \) rounds\)

Code for process \( p_i \)

\[
\begin{align*}
\text{pref} &= \text{my input} \\
&
\begin{align*}
\text{first round of phase } k, 1 \leq k \leq f+1: \\
&\text{send } \text{pref} \text{ to all} \\
&\text{receive } \text{pref’s of other processes} \\
&\text{pref} = \text{abort} \\
&\text{if some value } v \text{ appears } \geq n-f \text{ times then } \text{pref} = v
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{second round of phase } k, 1 \leq k \leq f+1: \\
&\text{send } \text{pref} \text{ to all} \\
&\text{receive } \text{pref’s of others} \\
&\text{if some value } v \text{ appears mult } > f \text{ times then } \\
&\text{pref} = \text{smallest such } v \\
&\text{// abort is largest}
\end{align*}
\]

\[
\begin{align*}
\text{third round of phase } k, 1 \leq k \leq f+1: \\
&\text{if } i = k \text{ then send } \text{pref} \text{ // I am phase queen} \\
&\text{receive tie-breaker from } k \\
&\text{if ( \text{pref} = \text{abort or mult } < n-f )} \\
&\text{and (tie-breaker } \neq \text{ abort)} \\
&\text{then } \text{pref} = \min(1, \text{tie-breaker})
\end{align*}
\]
Unanimous Phase Lemma

**Lemma:** If all nonfaulty processes prefer $v$ at start of phase $k$, then all prefer $v$ at end of phase $k$

For each phase $k$:

- At the end of the first round, the value of $\text{pref}$ for all nonfaulty processes, is $v$ or abort, for some $v \in \{0,1\}$
- At the end of the second round, the value of $\text{pref}$ for all nonfaulty processes, is $v$ or abort, for the same $v$

Nonfaulty Queen Lemma

**Lemma:** If $p_k$ is nonfaulty, then all nonfaulty processes have same preference at end of phase $k$

- All nonfaulty processes accept the phase king's message
- Some nonfaulty process ignores the king since $\text{mult} \geq n-t$. Then $\text{mult} > f$ for every nonfaulty process, and its $\text{pref}$ is the same.

After this phase, Unanimous Phase Lemma ensures agreement is maintained until the algorithm terminates
Randomized Consensus

- Weaken the termination condition and measure the expected time to termination
- Agreement and validity remain the same
- Allow to overcome the asynchronous impossibility and the synchronous lower bound (we’ll see only the first)

Two Sources of Nondeterminism

- In a **randomized algorithm**, processes flip coins to determine their next steps
  - Several possible executions
- But **even a deterministic algorithm** has several possible executions (from a fixed input)
  - Due to asynchrony and/or failures

- Separate the latter under the control of an **adversary**
  - Determines the next event to occur after an execution prefix
  - Must obey admissibility conditions according to model
  - May have other limitations (what information it can observe, how much computational power it has)
Evaluating a Distributed Randomized Algorithm

- An execution of a specific algorithm, \( \text{exec}(C_0, R, A) \), is uniquely determined by
  - an initial configuration \( C_0 \)
  - a sequence of random numbers \( R \)
  - an adversary \( A \)
- Given a predicate \( \text{Pred} \) on executions, a fixed adversary \( A \) and an initial configuration \( C_0 \)
  \[ \Pr[\text{Pred}] = \text{Prob} \{ R : \text{exec}(C_0, R, A) \text{ satisfies } \text{Pred} \} \]
- Let \( T \) be a random variable (time)
  \[ \exp(T, A, C_0) = \sum_t t \Pr[T = t] \]

Expected Time Complexity of a Randomized Distributed Algorithm

The **expected time complexity** is the max over all admissible adversaries \( A \) and initial configurations \( C_0 \), of the expected time for that particular \( A \) and \( C_0 \)

\[ \max_{\text{adversary } A, \text{ initial configuration } C_0} \exp(T(\text{Alg}, A, C_0)) \]

**Worst-case average:** for the worst adversary (asynchrony and failures) and initial configuration, average over the random choices of the algorithm

Extend naturally to other measures (like RMRs)
Structure of Consensus Algorithm

The algorithm has two components

• **Phase-based voting scheme** using individual processors’ preferences to reach agreement (when possible)
  We use [extended adopt-commit](#)

• A **shared coin procedure** used to break ties among these preferences

Shared Coin

A **shared coin** with [agreement probability](#) $\rho$ (with no input) returns a binary output, s.t.

- For every $v \in \{0,1\}$, all nonfaulty processors executing the procedure **output v with probability at least** $\rho$

A simple and very resilient [shared coin](#) with $\rho=1/2^n$ bias is when each process outputs a (uniform) random bit

There are more sophisticated constructions
Recall: Adopt-Commit

The decision is \((\text{grade}, y_i)\), where grade is either \text{adopt} or \text{commit}, such that

- **Graded agreement**: if a process decides \((\text{commit}, y_i)\) then all processes decide \((\text{adopt}, y_i)\) or \((\text{commit}, y_i)\)
- **Validity**: \(y_i\) was proposed by some process
- **Convergence**: If only \(y_i\) is proposed before \(p\) outputs \((\text{grade}, y_i)\) then grade = commit
- **Termination**: A process returns within a finite number of steps

---

Extended Adopt-Commit

The decision is either \(\bot\) (\text{abort}) or \((\text{grade}, y_i)\), where grade is either \text{adopt} or \text{commit}, such that

- **Graded agreement**: if a process decides \((\text{commit}, v)\) then all processes decide \((\text{adopt}, v)\) or \((\text{commit}, v)\) and if a process decides \((\text{adopt}, v)\) then no process adopts a different value
- **Validity**: if all nonfaulty processes propose \(v\) then all nonfaulty processes return \((\text{commit}, v)\)
- **Termination**: A process returns within a finite number of steps
Implementing Extended Adopt-Commit w/ Byzantine Failures

- Assumes $n > 3f$
- 2 (asynchronous) rounds

send $v$ to all
receive values from others
let $maj$ be value that occurs > $n/2$ times (0 if none)
let $mult$ be number of times $maj$ occurs
if $mult \geq n-f$ then send $maj$ to all

receive values from others
let $maj'$ be value that occurs most times
let $mult'$ be number of times $maj'$ occurs
if $mult' \geq n-f$ return $(commit,maj')$
else if $mult' \geq f+1$ return $(adopt,maj')$
else return abort

Randomized Consensus w/ Extended Adopt-Commit

Assume we have a shared coin algorithm with agreement probability $\rho$ and time complexity $T_{coin}$

pref = my input
Phase k
$(grade,v) = \text{Extended-adopt-commit}(pref)$
flip = Shared-coin()
if grade == abort
    pref = flip
if grade == adopt
    pref = v
else // grade == commit
    decide v // but continue to echo

Time complexity of a phase is $(T_{EAC} + T_{coin})$
Validity

Unanimous Phase Lemma: If all nonfaulty processes prefer $v$ at start of phase $k$, then all do at end of phase $k$

If all processes have input $v \Rightarrow$ all prefer $v$ in phase 1
By the lemma (and graded agreement),
all nonfaulty processes decide $v$ in phase 1

Agreement

Lemma: If $p_i$ decides $v$ in phase $r$, then all nonfaulty processes decide $v$ by phase $r + 1$

Proof: Let $r$ be the earliest phase in which a process (say, $p_i$) decides (say, on $v$)

$p_i$ got $(commit, v)$ in phase $r$

All other processes got $(adopt, v)$ in phase $r$, so they prefer $v$ in phase $r+1$ and by previous lemma, decide $v$
Termination

**Lemma:** The probability that all nonfaulty processes decide in a phase is at least $\rho$

**Proof:** If all nonfaulty processes set their preference in phase $r$ using Shared-coin
- With probability $2\rho$, they all get the same value ($\rho$ for 0 and $\rho$ for 1); lemma follows from unanimous phase lemma

If some processes **do not** set their preference using Shared-coin
- All of them have the same value $v$ as phase $r$ preference
- With probability $\geq \rho$, all processes get $v$ from Shared-coin

Expected Number of Phases

Probability of all deciding in any given phase $\geq \rho$
\[ \Rightarrow \text{Probability of terminating after } i \text{ phases is } (1-\rho)^{i-1}\rho \]
\[ \Rightarrow \text{Number of phases until termination is a geometric random variable whose expected value is } 1/\rho \]

The time complexity of the algorithm is $\rho^{-1}(T_{EAC}+T_{Coin})$, $T_{EAC}$ is the time complexity of Extended Adopt-Commit $T_{Coin}$ is the time complexity of Shared-coin
Better Shared Coin

Back to shared memory and crash failures...

- constant agreement probability $\rho$
- polynomial total number of steps $T_{\text{coin}}$

```
Shared SumCoins[i], NumFlips[i], initially 0
while ()
    c = random(-1,+1)
    SumCoins[i] += c // written only by i, atomic
    NumFlips[i]++ // written only by i, atomic
    read NumFlips[0,...,n-1]
    if $\Sigma$ NumFlips[0,...,n-1] > $n^2$
        read SumCoins[0,...,n-1]
        return( sign( $\Sigma$ SumCoins[0,...,n-1] ) )
```

Simple than $(0,1)$

Step Complexity

- Number of coins flipped (= iterations of the while loop) < $n^2+n$
- $O(n)$ steps per iteration $\Rightarrow O(n^3)$ total work

```
Shared SumCoins[i], NumFlips[i], initially 0
while ()
    c = random(-1,+1)
    SumCoins[i] += c // written only by i, atomic
    NumFlips[i]++ // written only by i, atomic
    read NumFlips[0,...,n-1]
    if $\Sigma$ NumFlips[0,...,n-1] > $n^2$
        read SumCoins[0,...,n-1]
        return( sign( $\Sigma$ SumCoins[0,...,n-1] ) )
```
Agreement Parameter

- Among $t^2+t$ independent unbiased coins, the minority is less than $t^2/2$ with probability $> 1/2$
- Probability all processes get same value $> 1/4$

```plaintext
Shared SumCoins[i], NumFlips[i], initially 0

while ()
    c = random(-1,+1)
    SumCoins[i] += c  // written only by i, atomic
    NumFlips[i]++  // written only by i, atomic
read NumFlips[0,...,n-1]
if Σ NumFlips[0,...,n-1] > n²
    read SumCoins[0,...,n-1]
return( sign( Σ SumCoins[0,...,n-1] ))
```

Space Lower Bound

$Ω(n)$ registers are necessary for nondeterministic solo-terminating consensus using reads and writes

[Leqi Zhu, 2016]

Use nondeterminism to capture randomization

- Multiple solo executions (of the same process) from a specific configuration

Proof very similar to $Ω(n)$ mutex space l.b.
Recall Valence w/ Small Twist

For a configuration C, and processes pi and pj

- pi is **v-solo-potent** in C if pi can decide v in some solo execution from C
- pi is **v-solo-univalent** in C if pi is v-solo-potent but not v-potent in C (solo-univalent in general)
- pi and pj are **solo-bivalent** in C if pi is v-solo-potent in C and pj is v-potent in C

\[ p_0 \text{ and } p_1 \text{ are solo-bivalent in some initial configuration} \]

Follows by a standard proof

Also, Recall Covering

A **process covers a register** R in a configuration C if it is about to write to R in C

Extend to a **set of k processes covering k registers**

**Block write**: all processes write

Assume processes P cover registers R in a configuration C.
Let C’ be the configuration after their block write from C,
and assume process p_j ≠ p_i is v-potent in C’. If a process p_i ∈ P, p_j ≠ p_i, is v-solo-potent in C,
then p_i writes to a register ≠ R in its solo execution
Proof by Contradiction

If $p_i$ writes only to $R$
Apply the solo execution of $p_j$
after the block write
$p_i$ still decides $\bar{v}$
$\Rightarrow$ Contradiction to agreement

Assume processes $P$ cover registers $R$ in a configuration $C$.
Let $C'$ be the configuration after their block write from $C$,
and assume process $p_j \neq p_i$ is $\bar{v}$-potent in $C'$.
If a process $p_i \notin P$, $p_j \neq p_i$ is $v$-solo-potent in $C$,
then $p_i$ writes to a register $\notin R$ in its solo execution

Key Lemma

If $p_0$ and $p_1$ are solo-bivalent in a configuration $C$ then
there is a {$p_0, \ldots, p_k$}-only execution from $C$ ending in $C'$
s.t. $p_0$ and $p_1$ are solo-bivalent in $C'$ and
$p_2, \ldots, p_k$ cover $k-1$ different registers in $C'$

Apply the lemma with $k = n - 1$,
starting from an initial configuration
in which $p_0$ and $p_1$ are solo-bivalent
$\Rightarrow n - 2$ space lower bound
Can be improved to $n - 1$
Help Lemma

If \( p_0 \) and \( p_1 \) are solo-bivalent in \( C \) and processes \( P \) (other than \( p_0 \) and \( p_1 \)) cover a set of registers \( R \), then \( p_0 \) and \( p_1 \) are solo-bivalent after a solo execution by either \( p_0 \) or \( p_1 \), followed by a block write to \( R \).

Proof of Help Lemma

If \( p_0 \) and \( p_1 \) are solo-bivalent in \( C \) and processes \( P \) (other than \( p_0 \) and \( p_1 \)) cover a set of registers \( R \), then \( p_0 \) and \( p_1 \) are solo-bivalent after a solo execution by either \( p_0 \) or \( p_1 \), followed by a block write to \( R \).
Proof of Help Lemma

Proof of Help Lemma

\[ p_0 \text{ and } p_1 \text{ are solo-bivalent} \]

\[ \text{block write to } R \]

\[ C' \]

\[ p_0 \text{ and } p_1 \text{ both v-solo-univalent} \]

\[ \text{Case 1: This step is a read or a write to a register } \in R \]

\[ p_1 \text{ does not distinguish } D \text{ and } D' \]

\[ \Rightarrow p_1 \text{ is v-solo-univalent in } D' \]

By maximality, \( p_0 \) is \( \bar{v} \)-solo-potent in \( D' \)

\[ \Rightarrow p_0 \text{ and } p_1 \text{ are solo-bivalent in } D' \]

\[ \checkmark \]
Proof of Help Lemma

Case 2: This step is a write to a register \( \not\in R \)
Commutativity \( \Rightarrow \) \( p_1 \) is \( v\)-solo-univalent in \( D' \)
By maximality, \( p_0 \) is \( v\)-solo-potent in \( D' \)
\( \Rightarrow \) \( p_0 \) and \( p_1 \) are solo-bivalent in \( D' \)

Back to Proving the Key Lemma

If \( p_0 \) and \( p_1 \) are solo-bivalent in a configuration \( C \) then there is a \( \{p_0, \ldots, p_k\}\)-only execution from \( C \) ending in \( C' \)
s.t. \( p_0 \) and \( p_1 \) are solo-bivalent in \( C' \) and \( p_2, \ldots, p_k \) cover \( k-1 \) different registers in \( C' \)

By induction on \( k \), with a trivial base case \( k=1 \)
For the induction step \((k+1)\)
Repeated apply induction hypothesis & help lemma
Back to Proving the Key Lemma

If $p_0$ and $p_1$ are solo-bivalent in a configuration $C$ then there is a $\{p_0, \ldots, p_k\}$-only execution from $C$ ending in $C'$ s.t. $p_0$ and $p_1$ are solo-bivalent in $C'$ and $p_2, \ldots, p_k$ cover $k-1$ different registers in $C'$.

Repeated apply induction hypothesis & help lemma

For some $i < j$, $R_i = R_j$

By first lemma, $p_{k+1}$ writes to a register.

Complete as in mutex lower bound
Finale

“At last we’ve reached a consensus!
This meeting is boring!”